A Git Tutorial: Commits, Branches, and Diffs

Nathan Green (UCL)

Commits, Diffs, and Tags

We can connect the fundamental concepts of Git to a typical data science workflow using three key ideas: the **repository**, the **commit**, and the **diff**.

Recall that a **repository** (or repo) is just a directory of files that Git manages holistically. A **commit** functions like a snapshot of all the files in the repo at a specific moment.

In fact, the notion of any specific version of a file is as an accumulation of diffs. If you go back far enough, you find the commit where the file was created in the first place. Every later version is stored by Git as that initial version, plus all the intervening diffs in the history that affect the file.

Every time you make a commit you must also write a short **commit message**. Ideally, this conveys the **motivation** for the change; the diff will show the **content** so don't talk about this. When you revisit a project after a break or need to digest recent changes made by a colleague, looking at the history, by reading commit messages and skimming through diffs, is an extremely efficient way to get up to speed.

Every commit needs some sort of ID and Git does this automatically, assigning each commit what is called a **SHA**, a seemingly random string of 40 letters and numbers. Usually the first 7 characters suffice. You can also designate certain snapshots as special with a **tag**, which is a name of your choosing. In a software project, it is typical to tag a release with its version, e.g., "v1.0.3". For a manuscript or analytical project, you might tag the version submitted to a journal or transmitted to external collaborators.

Basic Commands

Create a repo

Create a new folder in Explorer or with

```
mkdir git-practice
```

Move to folder in Explorer or with

```
cd git-practice
```

Create a new repository with

```
git init
```

You'll now see a `.git` folder in your repo folder. This is the only difference to how you would work otherwise. This folder contains the notes of the changes made with each commit.

Creating a Dummy File

You'll need to create empty files to practice commands.

Use your File Explorer (GUI) by right-clicking within a folder. The steps are very similar across Windows, macOS, and Linux.

1. Navigate to the project folder where you want to create the file.
2. **Right-click** on an empty space inside the folder.
3. From the context menu, select **New**, then choose **Text Document** (Windows) or an equivalent option like **New File** (macOS/Linux).
4. Name the new file **foo.txt** and press Enter. Make sure the extension is `.txt` and not something like `.txt.txt`.

or use the command line. The fastest way to create an empty file is with the `touch` command in your terminal. This command updates the modification time of an existing file or creates a new, empty file if it doesn't exist.

1. Open your terminal (like Terminal, Git Bash, or PowerShell).
2. Navigate to your project directory using the `cd` command.
3. Run the following command:

```
"hello" > foo.txt
```

This will instantly create an empty file named **foo.txt** in your current directory. You can confirm it was created by running the `ls` command to list the contents of the directory.

Stage and Commit Local Changes

Use `git add` to stage files (i.e., select them to be included in the next commit) and `git commit` to save the snapshot.

```
# Stage a specific file for the next commit
git add foo.txt

# Commit the staged changes with a descriptive message
git commit --message "A commit message"
```

Check the State of the Repository

These commands help you see the current status, untracked files, and the history of commits.

```
# See the status of your working directory and staging area
git status

# View the detailed commit history
git log

# View a condensed, one-line version of the commit history
git log --oneline
```

Compare Versions

Use `git diff` to see the exact changes between commits or between your current work and the last commit.

```
# View differences between your working directory and the last commit
git diff
```

Practice Recovering From Mistakes

This section covers how to recover from common mistakes. You can practice these commands in any local Git repository you've created. These operations do not involve GitHub.

A word of caution: if the commit you want to fix is not your most recent one, seriously consider just letting it go. Trying to change older history can get complicated quickly.

Undoing the Last Commit with `git reset`

So, you want to undo the last commit? You have a few options depending on what you want to do with the work you've committed. The `HEAD^` syntax refers to the commit *before* the most recent one.

Option 1: Discard Everything (Hard Reset)

This option completely undoes the last commit and **discards all of the changes** in those files. Use this with caution!

- **Use Case**: You want to completely erase the last commit and any work associated with it.
- **Command**: `git reset --hard HEAD^`
- **Outcome**: You will lose any changes that were not reflected in the commit-before-last. Your project is now in the exact state it was in before you made the bad commit.

```
# WARNING: This deletes your work from the last commit
git reset --hard HEAD^
```

Option 2: Keep Changes, Unstage Files (Mixed Reset)

This is the default reset mode. It undoes the commit but leaves the files in your working directory as they were. The changes will be unstaged.

- **Use Case**: You like the changes you made, but you want to recommit them differently, perhaps in multiple, smaller commits.
- **Command**: `git reset HEAD^`
- **Outcome**: The commit is undone. Your files still contain all the changes, but they are no longer staged for the next commit.

```
# This is the same as `git reset --mixed HEAD^`
git reset HEAD^
```

Option 3: Keep Changes, Keep Files Staged (Soft Reset)

This option is the least destructive. It undoes the commit but leaves your working directory and your staging area exactly as they were.

- **Use Case**: You just want to edit the commit message or add one more small change to the last commit.
- **Command**: `git reset --soft HEAD^`
- **Outcome**: It's as if you ran `git add` but never ran `git commit`. You are right back to the moment before you committed.

```
# Undoes the commit but leaves everything staged
git reset --soft HEAD^
```

Amending the Most Recent Commit

If you just want to tweak the most recent commit—like fixing a typo in the commit message or adding a file you forgot—it's often easier to **amend** it rather than resetting.

First, make any changes you need to your files and stage them with `git add`. If you only want to change the commit message, you don't need to change any files.

To amend from the command line, you can either have Git open an editor to let you create the new message or provide the new message directly.

```
# This opens your default text editor to change the commit message
git commit --amend
```

```
# This amends the commit and provides a new message directly
git commit --amend -m "New, corrected commit message"
```

Difference between `git add .` and `git commit -all`

In Git, committing is a two-step process: first you *stage* changes with `git add`, then you *commit* them with `git commit`. Two common workflows accomplish this, but they behave differently, especially concerning new files.

The key difference between `git commit -a` and `git add .` is how they handle **new, untracked files**.

The `git commit -a` command (or `git commit --all`) is a shortcut that combines staging and committing into a single step. However, it only stages files that Git is already tracking.

- **What it does**: Automatically stages all modified or deleted files that have been previously committed to the repository, and then opens the commit message editor.
- **What it ignores**: It will completely ignore any **new files** you have created that Git does not yet track.

The comprehensive method is to use `git add .` and `git commit`. This is the standard, explicit two-step workflow. The `git add .` command is a powerful staging tool.

- **What it does**: `git add .` stages **all** changes in the current directory. This includes modifications to tracked files, deletions of tracked files, and **any new, untracked files**.
- **What it ignores**: Nothing. It stages everything.